Int J Parallel Prog (2016) 44:233–256
DOI 10.1007/s10766-015-0353-x

SCnC: Efficient Unification of Streaming with Dynamic Task Parallelism

Dragoş Sbîrlea · Jun Shirako · Ryan Newton · Vivek Sarkar

Received: 28 February 2013 / Accepted: 19 February 2015 / Published online: 3 March 2015
© Springer Science+Business Media New York 2015

Abstract Stream processing is a special form of the dataflow execution model that offers extensive opportunities for optimization and automatic parallelization. To take full advantage of the paradigm programmers are typically required to learn a new language and re-implement their applications. This work shows that it is possible to exploit streaming as a safe and automatic optimization of a more general dataflow-based model—one in which computation kernels are written in standard, general-purpose languages and organized as a coordination graph. We propose streaming concurrent collections (SCnC), a streaming system that can efficiently run a subset of programs supported by concurrent collections (CnC). CnC is a general purpose parallel programming paradigm that integrates task parallelism and dataflow computing. The proposed streaming support allows application developers to reason about their program as a general dataflow graph, while benefiting from the performance and tight memory footprint of stream parallelism when their program satisfies streaming constraints. In this paper, we formally define the application requirements for using SCnC, and outline a static decision procedure for identifying and processing eligible SCnC subgraphs. We present initial results showing that transitioning from general CnC to SCnC leads to a throughput increase of up to 40× for certain benchmarks, and

D. Sbîrlea (✉) · J. Shirako · V. Sarkar
Rice University, 6100 Main St, Houston, TX 77030, USA
e-mail: dragos@rice.edu; sbarlea@gmail.com

J. Shirako
e-mail: shirako@rice.edu

V. Sarkar
e-mail: vsarkar@rice.edu

R. Newton
Indiana University, 107 South Indiana Avenue, Bloomington, IN 47405, USA
e-mail: rrnewton@indiana.edu

also enables programs with large data sizes to execute in available memory for cases where CnC execution may run out of memory.

Keywords Streaming · Task parallelism · Dynamic parallelism · Dataflow

1 Introduction

As multicore computing becomes the norm, exploiting parallelism in everyday applications is an important concern. Parallel programming models based on stream processing are of particular interest because they offer some of the best examples of high-performance, fully automatic parallelization of implicitly parallel code. However, programming languages built around stream programming, such as StreamIt [1], have not become widespread, even for application areas such as digital signal processing where they offer clear benefits. A possible reason for this is the intrinsic adoption barrier faced by new languages—this is especially a problem since today's streaming languages often suffer from a narrow range of applicability and inability to compose with larger software systems.

We propose a new solution to this problem that can enable efficient stream processing within more general applications written using standard languages (in this paper, Java). We do not, however, attempt to optimize streaming patterns within a fully general-purpose language. Such an approach is possible, with specialized streaming libraries, but suffers from the same drawbacks of stream programming languages: high adoption barriers and the need for programmers to be aware that their application is streaming. Instead, we advocate the use of the compiler to target stream-processing patterns automatically located within a applications written for a general-purpose graph-based parallel model such as *concurrent collections* [2].

Concurrent collections (CnC) is a dataflow language with a very dynamic nature which makes it compatible with expressing a large set of applications. CnC allows the user to dynamically launch tasks ("steps") which read and write from key-value stores ("collections") which obey a dynamic single assignment rule. CnC is an effective parallel programming model for many problems, but for streaming applications, as with other task parallel models, it is rather inefficient: it incurs task scheduler overhead and has further overhead to store stream elements in heavy-weight general-purpose collections.

In this paper we introduce Streaming CnC (SCnC), a streaming system which identifies streaming patterns based on metadata already available in CnC programs. For compliant programs, SCnC uses an alternative code generator and streaming runtime which improves the performance and memory usage of streaming applications relative to standard CnC.

The main contributions of this paper are the following:

– The identification of a subset of a general task-parallel programming model which makes feasible the compile-time identification of streaming patterns (in Sect. 3). The recognized streaming patterns are more general than the usual pipeline/split-join/loopback streaming graphs.

- A static analysis algorithm (Sects. 3.2 and 5) to identify when a program is streaming (i.e. will run correctly on the streaming runtime), as well as algorithmic transformation to bring it to such a form, if possible (Sect. 4).
- A static analysis algorithm to determine safe bounds for stream buffers, ensuring deadlock-free execution of streaming programs if the initial program is deadlock-free (described in Sect. 5.1). Together, the static analysis algorithms ensure that the streaming and the task-parallel executions of compliant programs are equivalent.
- A feature that identifies opportunities for converting stream data accesses to local state accesses (Sect. 6.1). The support for dynamic parallelism (Sect. 6.2) integrates well with the local state access, enabling each dynamic filter to have its own local state accessible between iterations.[1] To our knowledge, no other streaming system enables this.
- Experimental results on our implementation of SCnC show compelling performance and memory improvements compared to task-based execution of the same applications.

2 CnC Background

This section briefly reviews concurrent collections (CnC) [2], the programming model which forms the basis for streaming concurrent collections. CnC programs are composed of three types of constructs: item collections, control collections and step collections. These collections and their relationships are defined statically for each application in a CnC graph specification file; the code of the application can be written in any one of multiple host languages for which CnC has a runtime available. The main program that initiates a CnC graph, provides its inputs, and reads its outputs is referred to as the environment, and appears as a special node in the CnC graph.

Step collections group together dynamic instances ("step instances" or simply "steps") of tasks. A step instance is "prescribed", analogous to a task being spawned or a streaming filter performing an iteration). Steps are stateless and have no side effects outside of producing and consuming items. In contrast to streaming, the CnC model does not imply any ordering constraints between the execution of step instances of the same step collection; however, step instances read and write data values called items which introduce dataflow dependencies between them.

Item collections are CnC's data layer and accessing items is the sole means of synchronization between steps. Item collections are "append-only" key-value stores with the keys referred to as item keys. Any step collection that shares an edge with an item collection in the application graph can perform a `put(key, value)` or `get(key)` to read and respectively write items in that collection, according to the direction of the edge. Each individual item is dynamic-single-assignment and its value cannot be overwritten. A `get` on an unavailable item requires deferring the consumer step until the item is `put` by another step instance.

[1] The OpenStream system, developed concurrently with this work, has a similar feature, but the OpenStream state cannot be inferred from stream accesses.

Table 1 Types of edges in CnC graphs

Edge type	Source col.	Destination col.	Meaning
Item-put	Step	Item	A source step may `put` items in the destination item collection
Control-put	Step	Control	A source step instance may `put` control tags in the destination collection
Item-get	Item	Step	A destination step may `get` items from the source collection
Prescription	Control	Step	Any tag `put` in the source control collection leads to the execution of a step instance from the destination collection

With step collections used for computation and item collections storing data, the final collection type, control collections, serve as broadcast nodes, sending invocation messages (also called control tags) to one or more step collections. These messages prescribe the execution of a new step instance of receiving step collection. The operation on control collections through which this is done is `put(controlTag)`.

The CnC graph specification is a textual representation of the statically known structure of the application, relating the application's step, item, and control collections, as well as including metadata relating the keys of items accessed to the control tag of the step. The graph is used to generate code for the item and control collections and to construct and execute the graph, so that the user need write only step implementations. For graphical representations of CnC graphs, we use the following shapes, following standard CnC conventions: step collections are circles, item collections are squares and control collections are triangles. We classify the various types of edges of a CnC graph as shown in Table 1.

It supports a wide range of application graphs and provably supports more parallel execution graphs than the popular task-parallel language, Cilk: while Cilk supports fully strict computations[3] that are also terminally strict [4], CnC can express any terminally strict computation [5]. Compared to other dataflow languages, CnC allows individual tasks to interact with a dynamic number of other tasks (the communication pattern are decided when each tasks start running) and a dynamic number of times, making it much more flexible than other dataflow models.

3 Streaming Concurrent Collections

Streaming concurrent collections (SCnC) provides a restricted version of the CnC model (including the graph specification, corresponding code generator, and runtime library) that enables efficient streaming execution of compliant CnC programs, as opposed to task-based execution used for non-streaming CnC applications. Ultimately, SCnC will execute CnC applications using the same source code and specifications as CnC, but our current prototype has minor API differences, described in Sect. 7.

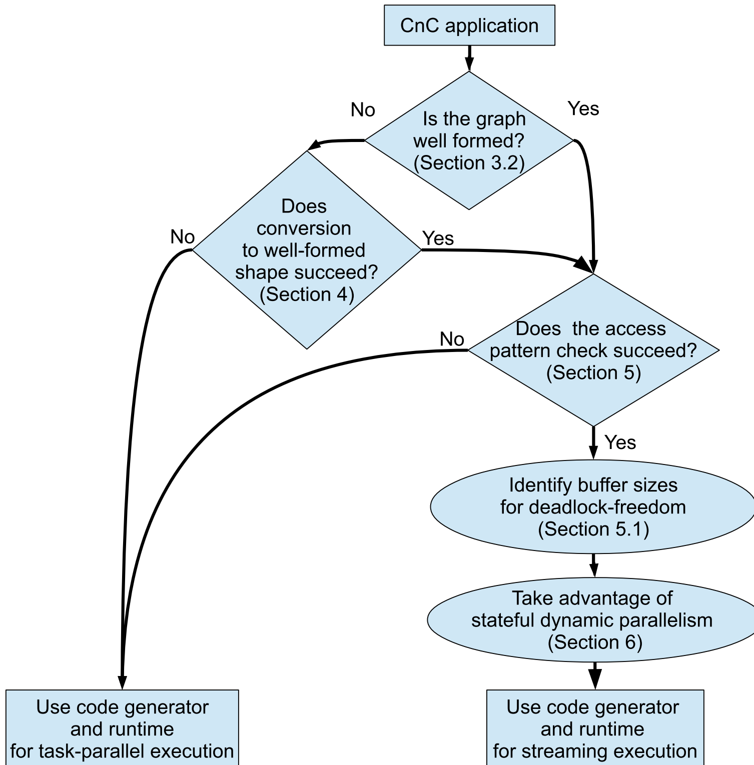

Fig. 1 The sequence of tests performed by SCnC to choose the appropriate runtime for each application

In streaming (SCnC) execution (compared to dataflow task execution) the efficiency increase comes mainly from two sources: lower task overhead (for streaming programs, many short lived tasks can be replaced by fewer long lived tasks) and lower data access costs (possible due to the more predictable data access patterns in streaming).

3.1 SCnC Workflow

Not any CnC application is a streaming application because streaming has restrictions of both the shape of the application graph and on the data access patterns of the application code. To make sure that we only use the streaming runtime if the application is compatible with it, streaming compliance is validated through a sequence of compile-time tests illustrated in Fig. 1.

First, the shape of the application graph must have some characteristics that we call well-formed [for SCnC] described in Sect. 3.2. Most of these characteristics are needed to ensure that the application is streaming, but some are engineering considerations, as presented in Sect. 8. If the test fails because the application graph is not well-formed, then we attempt to convert it to a well-formed shape, as discussed in Sect. 4; if the conversion fails, we fall back to the task-parallel runtime. If it succeeds, additional

tests must be performed on the item accesses performed by each step, to ensure they respect the limitations expected from a streaming program, such as the limited lifetime of items. The data access tests are described in Sect. 5. Once these tests passes, we size the streaming buffers to the minimum size that is proven to ensure deadlock-freedom. Finally, the task-based runtime is switched to the streaming runtime, using the safety bounds inferred above to instantiate all relevant buffers.

As reflected in the figure, our current prototype makes an all-or-nothing decision as to whether an application is streaming, forcing the user to explicitly factor out streaming portions of their application into separate CnC specifications. However, there is no reason that SCnC sub-programs cannot be automatically extracted in the future.

3.2 SCnC Well-Formed Graphs

The SCnC runtime cannot run an arbitrary CnC graph. For a CnC program, the *well-formed* graph shape guarantees that, if the application also respects the data access patterns discussed in Sect. 5, then streaming execution is possible and will be deadlock free (see also Sect. 5.1).

Definition A *well-formed* SCnC graph is a CnC graph that respects the following conditions:

1. Control collections have only one step collection that performs `put` operations in it and only one prescribed step collection.
2. Item collections have only one step collection that performs `put` and only one that performs `get`.
3. The environment only `puts` control tags into a single control collection and has no item-put edge. The control tags can be pairs that include the value of any items, or the entry step itself may be the source of items.

Definition The CnC control graph is the CnC graph with the item collections, item put-edges and item get-edges removed.

Theorem 1 *For a well formed CnC graph, the CnC control graph rooted at the entry control collection is a directed tree.*

Proof We first prove the absence of cycles, relying on the following: the CnC control graph is weakly connected, both step and control collections have only one predecessor and can be reached from the environment. If there was a cycle, the nodes in that cycle would have a predecessor in the cycle, so either the environment is in the cycle (impossible, as the environment does not have any incoming edges) or the cycle is unreachable from the environment (impossible). Because it does not have cycles, the CnC control graph rooted in the entry control collection is a DAG. Further, because of the constraint on the in-degree of control collections, the DAG must be a tree.

We can safely assume that there is a path from the environment to each step collection (otherwise these unreachable step collection will never lead to steps being executed, so they are dead code). As the entry control collection is the singular child

of the environment, all paths must pass through it, so there must be a path from the entry control collection to every node.

We now check for the proper direction of the edges: CnC control graph edges are either from control collections to step collections or from step collections to control collections. Each step collection has only one incoming edge (the prescription edge) which connects the step collection to its parent in the tree, leaving the ether edges to connect the step collection to children. Thus the CnC control graph rooted in the entry control collection is a directed tree, with all edges transferring control away from the environment. □

An intuition about the generality of the graph shapes allowed by SCnC is as follows. Any directed tree can be a CnC control graph (if we imagine that each node is replaced by a control/step-collection pair). On this directed tree backbone, adding any number of item collections (streams) between step collections is permitted, as long as they remain single-source/single-destination. This last restriction is relaxed in the following section.

4 Converting CnC Graphs to Well-Formed Shape

In some cases, graphs that are not well-formed because they do not observe the single producer or single consumer rules can be rewritten to a well-formed shape by applying a sequence of transformations, as follows:

1. First, we reshape the environment node to only `put` control tags in a single control collection. We add a new step collection and control collection for interaction with the environment. This step collection (called the *entry* step collection) will serve as source for the items that would have been `put` from the environment. Figure 2 illustrates this transformation.
2. We can then turn multi-prescription control collections to single-prescription control collections, by splitting each multi-prescription control collection into multiple control collections, as shown in Fig. 3. The producing step will now need to perform a `put` with the same control tag that was used in the initial implementation, but the `put` will be performed multiple times, once in each new control collection.
3. Next we eliminate any multiple-producer item collections by splitting the item collection, as shown in Fig. 4. We also add a new step collection, prescribed by one of the original producers, that functions as a custom join step: it gets items from all split collections and `puts` them into a single result item collection. The join step must be able to decide, based on its control tag, what item collection and what item key it should `get` and then `put` in the merged item collection. This is easy when the producer steps populate the item collections in a consistent pattern, such as round robin, but become difficult if this is not the case, requiring the user to write custom code for it. The step collection will need to `put` items in the merged item collection with the same key that they were indexed in their source item collections. The single-assignment rule of CnC ensures that there will not be multiple items with the same key in the split item collection, as all of them were in a single item collection in the initial CnC program.

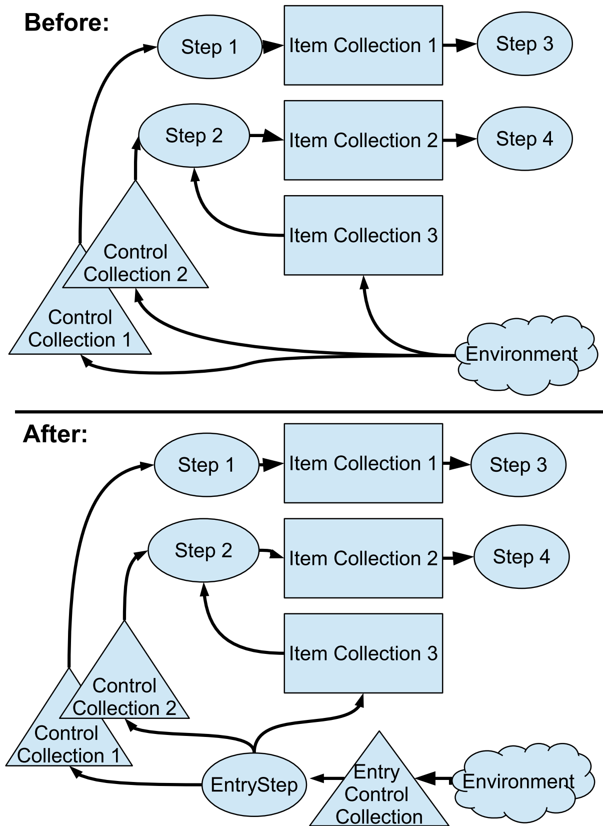

Fig. 2 Conversion of the environment from multiple-producer to single-producer

4. To remove multiple-consumer item collections, we convert them into single-consumer item collection. To do this, we split the item collection into N separate item collections, one for each consumer, as shown in Fig. 5. All `puts` that were initially performed to the multiple-consumer item collection should now happen N times, once for each split item collection. These `put` calls should use the same item keys they used in the initial CnC program.

After these transformations have been performed, the resulting graph, even if well-formed, may have data access patterns that are incompatible with streaming. The following section proposes a static analysis approach of detecting if an application complies with streaming access pattern restrictions.

5 Streaming Access Pattern Identification

Because the streaming runtime is more restricted in its use of items than CnC, additional checks have to be performed before using it. In this section we present an algorithm to perform the required checks for automatic identification of the streaming

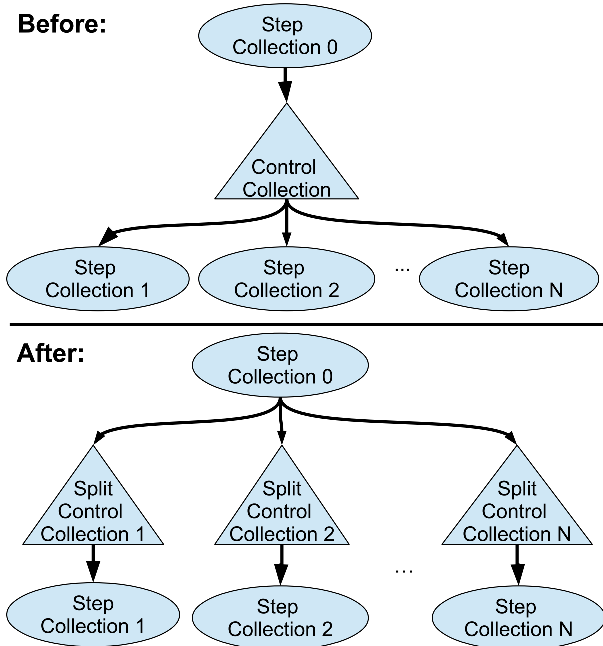

Fig. 3 Conversion of a control collection from multiple prescribed steps to a single prescribed step

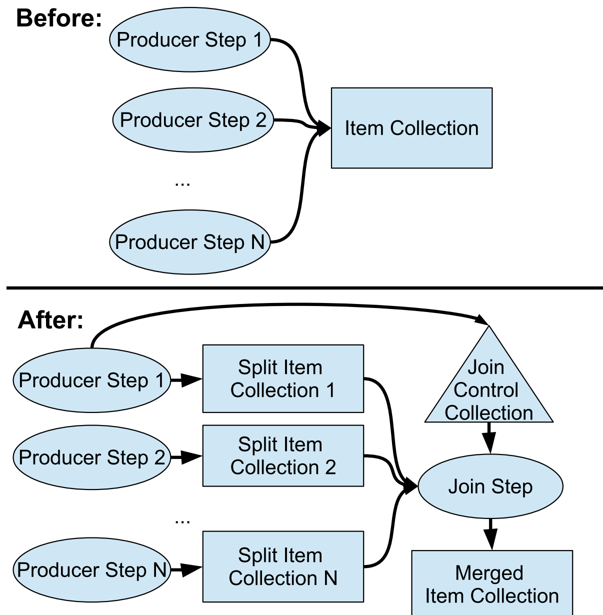

Fig. 4 Conversion of an item collection from multiple-producer to single-producer

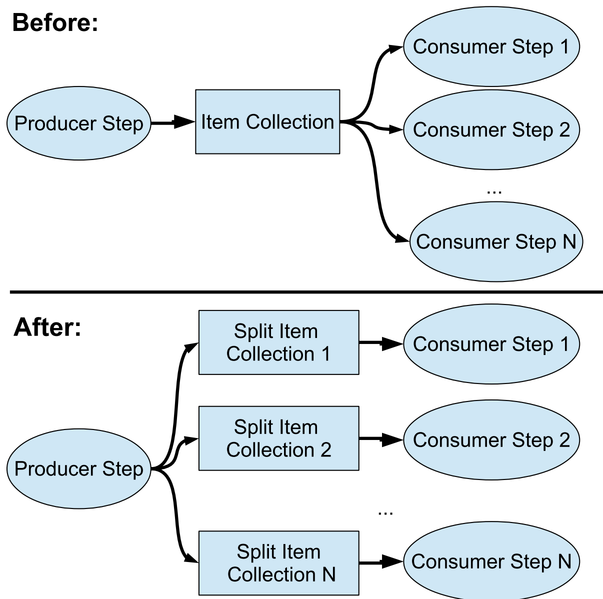

Fig. 5 Conversion of an item collection from multiple-consumers to single-consumer

access patterns on a well-formed CnC graph. The output of the algorithm is a boolean value that answers the question: would the streaming runtime run the application correctly? The algorithm relies on the the computation of mathematical functions, but sometimes such functions cannot be identified (which happens, for example, if the function output does not dependent only on the parameters specified as input). In that case, the algorithm outputs FALSE, meaning that the application cannot be executed using the SCnC runtime. These functions that annotate edges in this algorithm can be user written in CnC. The algorithms presented in this section have not been implemented, but automatic analysis could be performed by expressing the algorithm equations as mathematical constraints in any modelling tool which can compute the inverse of functions, conditions for function equality, etc. Such mathematical modelling has now made its way into commercial tools like Matlab [6]. For CnC applications, these functions are typically not complicated(usually they are linear functions), which considerably simplifies analysis. If mathematical modelling engine is not powerful enough to express all equations for all functions, it is always safe to fall-back to the task-based runtime.

The algorithm consists of two parts: graph analysis and condition testing. The intuition behind the algorithm is the following: every time a control tag is `put` from the environment, it starts a "wave" of step instances that prescribe other step instances throughout the graph. Some of these steps will `get` items, others with `put` them. The starting of these waves are independent from one another, as they come from the environment. By modelling the sets of items read and written by each such wave through mathematical functions, the algorithm identifies conditions under which streaming execution always succeeds; one condition (sufficient, but

very restrictive) is that waves only consume items produced in their previous wave and the streaming buffers are large enough to hold all items produced consecutive waves.

The analysis phase computes meta-information in the form of mathematical functions labelling the edges of the application graph. It consists of the following steps:

1. Ensure that all item keys and control tags have an integer ordering component, whose values are consecutive increasing integers. In the following steps of the algorithm, for simplicity, we refer only to this ordering component as the control tag or item key, even though there may be other information used in the real tag. We call the ordering component of the control collection that is produced by the environment iteration number.
2. Annotate graph edges with functions, as follows:
   - Annotate each item-put edge (from a step collection to an item collection) with one or more item-put functions with domain equal to the possible control tags of the step and co-domain equal to the item keys that are `put` . If a step instance can `put` k items, there have to be k distinct item-put functions, each modelling the relation between the tag of the step and the key of an item.

     Failure to correctly identify all item-put functions leads to the application being declared incompatible with the streaming runtime; this can happen if the maximum number of `puts` performed by each step instance is not bounded or if the functions cannot be identified statically (for example, when the item keys depend on the value of other data items).
   - Annotate each control-put edge with similar control-put functions, relating the control tag of the step to the values of the control tags it produces.
   - Annotate each item get-edge with similar item-get functions ($f_{itemGet}$), relating the control tag of the step to the values of the keys of items it produces.
   - Label each prescription edge with the identity function.
3. Compute functions that map iteration numbers to the item keys and control tags used throughout the graph.
   - Do a traversal of the CnC control graph (see definition 3.2), labelling each control-put and item-put edge with the composition of the functions through which the path from the root of the graph to that control/item collection. We call this function a producer function because it maps iteration numbers to the keys and tags produced during the CnC execution. If any edge has multiple item-put, item-get or control-put function annotations, there will be multiple producer functions. Given an input iteration, each of these functions describes one item key or control tag that was created because of that iteration.

     As an example of computing the producer functions, if the path from the entry control collection to a step collection S prescribed by control collection C goes through the edges labelled with the control-put functions f_1 to f_n, the producer function for that control collection is: $f^C_{producer} = f_n \circ f_{n-1} \circ f_{n-2} \circ \cdots \circ f_1$. If the step collection S produces items in the item collection I, and the item-put edge between them is labelled with item-put function $f_{itemPut}$, then the producer function for item collection I is: $f^I_{producer} = f_{itemPut} \circ f^C_{controlPut}$

   - In addition to creating producer function functions relating iterations to items produced, we create consumer functions for each key consumed. That is, we compose the producer function of steps with their item-get functions.
4. For each item-get edge in the graph, compute the minimum-consumer function, defined as the minimum of the values of all the consumer functions labelling that edge. Minimum-consumer function identify the smallest key that a step can `get` from a particular item collection, as a function of the iteration numbers.
5. Label item-put edges with minimum-producer functions computed similarly to minimum-consumer functions.

The test phase uses the above producer and consumer functions to decide if the item access patterns of the application respect the streaming access patterns. We express these streaming patterns as five constraints that, if followed, enable CnC application to run as SCnC applications. They are:

1. The *"producer precedence"* constraint: Any item is produced in an earlier iteration than that in which it is consumed. If $f_{consumer}$ and $f_{producer}$ annotate edges that share a common item collection vertex and y is an item key, then $f^{-1}_{producer}(y) \leq f^{-1}_{consumer}(y), \forall y \geq 0$. If there is no inverse for either functions of any item collection, return FALSE.
2. The *"bounded buffer"* constraint: There is a constant N such that for any iteration, the the ordering component of any two items consumed from each item collection in any iteration is not more than N. The corresponding equation is: $|(f_{consumer1} - f_{consumer2})(x)| < N, \forall x \quad and \quad \forall f_{cconsumer1}, f_{consumer2}$ consumer functions of a single step collection.
3. The *"sliding window"* constraint: For any consecutive step instances of a step collection, the minimum ordering component that can be consumed by the step second step is no lower than that that consumed by first step: $f_{min_consumer}(y) \leq f_{min_consumer}(y+1)$.
4. The *"bounded lifetime"* constraint: For any item tagged t, produced in iteration t_1 and consumed in iteration t_2, there is N_2 constant such that $t_2 - t_1 < N_2$. Note that individual consumer functions do not need to have this monotonicity property. Bounded buffer, sliding window and bounded lifetime assure that we will not need a buffer size larger than N_1 or N_2 to satisfy `get` calls on an item collection.
5. The *"unique timestep"* constraint: Each step instance performs no more than a single `put` in each of its output control collections. This constraint assures us that, for a given step collection there will never be more than one step instance with the same iteration number (started by a single ancestor).

5.1 Deadlock Freedom

To ensure that the SCnC execution is equivalent to the CnC execution of the same program, SCnC must not introduce deadlocks in correct CnC programs (correct programs are those in which no step is prescribed, but its inputs are never produced). There are three possible causes of deadlock in SCnC, as follows:

"Producer-consumer iteration ordering" deadlock. If a step blocks attempting to get an item that cannot be produced because it requires the current step to complete. This cannot happen with SCnC as it contradicts the "producer precedence" rule.
"Evicted item" deadlock. If a step performs a get on an item that is no longer in the streaming buffer (streaming access pattern constraint). This can happen only if the streaming buffer size is too small, which is prevented by the buffer sizing method described in the next Sect. (5.2).
"Full-empty buffer" deadlock [7,8]. This problem can appear, in its simplest form, if two steps are connected by two item collections A and B. If the buffer of A becomes full, this blocks the producer and prevents him from producing items in another queue B. If the consumer also blocks, because B is empty, it cannot unblock the producer by consuming items from A, then this deadlock is a full-empty buffer deadlock (A is full and B is empty). In the following Sect. (5.2), we describe a technique that statically finds a bound for the streaming buffers such that they never fill up.

5.2 Sizing the Stream Buffers

This subsection describes how to statically identify a safe size of the streaming buffers such that "full-empty buffer" and "evicted item" deadlocks can never manifest. First, it is important to notice that a program having only control and step collections cannot deadlock, as the control graph is always a tree.

For an item with key `t` produced within a well-formed CnC application we have the following equation: $t = f^{i}_{producer}(it_1) = f^{j}_{consumer}(it_2)$ illustrating that the item was produced in time iteration it_1 and consumed in time iteration it_2.[2] The required buffer size for item `t` is $(it_2 - it_1) * \max_{t=it_1..it_2} POR(t)$, where POR(it) is the producer output rate of iteration `it`. The item rate is upper-bound the cardinality C of the set of item-put functions labelling the item-put edge of the item collection. Also, according to the "bounded lifetime" constraint, there is an integer constant k such that $it_2 - it_1 < k$ which means that the items consumed by a step are produced a fixed number of time-steps before.

The item collection buffer size is thus upper-bound by $k * C$. If the actual buffer size of the item collection buffer is larger than this, the buffer will never fill, so the producer and consumer edges cannot participate in a deadlock cycle.

This condition is not sufficient though, as there might not be space in the control collection buffers somewhere on the path between the producer and the consumer.[3] To find an upper-bound for the size of the buffers on this path, we use the fact that each step can `put` at most one control tag per iteration in each control collection ("unique timestamp" rule). The maximum number of tags that need storage is thus $M = it_2 - it_1$; this limit applies for all control collections on the path between the producer and consumer steps.

[2] If there are multiple consumer functions, all combinations must be considered and only the maximum buffer size obtained is safe.

[3] As shown in Sect. 3.2, the control graph is a tree, so there is only one such path.

Fig. 6 The CnC step-local item collection pattern

As shown, only the combination of using sufficiently large buffers for item collections and control buffers ensures that SCnC execution does not introduce new deadlocks.

6 Optimizing SCnC Performance

Because CnC is optimized for task-based execution, ample opportunities for performance optimization are available if they are to be executed on a streaming runtime. In the following subsections, we describe two such optimizations.

6.1 Stateful Streaming Steps

In CnC, steps are stateless (interaction between step iteration cannot happen though step-local data), which enables task execution through an abort-restart mechanism; in SCnC, this restriction loses its motivation, as SCnC has a different model, in which long lived tasks execute all instances of each step collections. These long running tasks could store state that could be shared between iterations, but the stateless CnC program used as input makes such an approach challenging. We discuss two possible approaches to turn stateless steps into stateful steps below.

The first approach is based on taking advantage of patterns in the application graph. For example, one such pattern is the item collection/step collection cycle (shown in Fig. 6) in which a single step collection is both producer and consumer of an item collection. For well-formed graph the step collection, being the single producer and consumer, is the single entity to interact with the item collection. We call such item collections step-local item collections. The cause of this pattern is the CnC restriction that steps are stateless (that is, there is not state information preserved between different step instance executions). If using SCnC, these collections can optimized out by transforming back into data fields of the step collection. By doing so, we remove the synchronization overhead implied by the item collection.

This approach has the advantage of never impeding scalability because it only decreases overhead for existing synchronization.

The second approach is to allow alternate stateful implementations for CnC steps, used only by the SCnC. Compared to the pattern-based approach described above, this approach loses any potential of automatic conversion, but can lead to a larger performance improvement because it exploits more opportunities for local state conversion. This approach may lead to worse scalability because stateless steps are more difficult to run in parallel, but also has a higher potential for performance because programmers can exploit more opportunities for using state instead of items.

6.2 Dynamic Parallelism

Some streaming applications, such as compression/decompression and event monitoring [9], have time-varying data-rates and compute loads, depending on the phase of the computation or the availability of external inputs. Efficiently supporting these situation requires dynamically adjusting the parallelism used by the runtime.

The SCnC dynamic parallelism support is based on dynamically-parallel step collections. In the statically-parallel SCnC model, every step collection corresponds to a sequence of serial execution of step instances. Parallelism is gained by concurrently processing step instances of different item collections. With dynamically parallel step collections, a dynamic number of step instances of each item collection can execute in parallel as each step collection has a varying number of processing queues. Each step instance is assigned to exactly one of these. The SCnC `put(controlTag)` operation has optional integer parameter which identifies which queue of the prescribed step collection should process the control tag. When an identifier value is used for the first time, a new queue is instantiated and added inside the step collection, turning the step collection into a dynamic split-join node. Each processing queue maintains its own local state that can be used across step instances.

For stateless step collections, a compiler transformation can automatically distribute the step instances to the number of clones that yields the best performance in a round-robin fashion. However, applying the same approach for stateful steps is conditional on the possibility of separating the state in several independent queues and managing the state independently. This is very challenging task to automate and is not attempted by any of the current streaming systems. We take an intermediate approach, by looking for a dimension of the control tag which the programmer used to control the dynamic parallelism and distribution. We find potential such dimensions by inspecting the control tags of each step collections. If the control tags are two dimensional, after removing the ordering component, we check if the remaining dimension can be used as the processing queue id. A static analysis can test if the value of this dimension is included as a dimension in the item keys used by the step. If steps only read or write items whose keys have the same id, then the steps can safely be assigned to independent processing queues based on the value of this dimension. When parallelism needs to be reduced or queue state is no longer required, queues can be cleaned up by sending special control tags with the corresponding processing queue id. Note that the mechanism of using a dimension of the control tag to differentiate items is natural and used even in regular CnC, the only obstacle against automating this approach is if static analysis is sufficient to identify if steps only access items with he same queue id. In the applications we looked at, a simple analysis like copy propagation was enough.

This approach is high-level and does not require user intervention if the original CnC program uses the common CnC pattern of separate tag dimension for encoding the state. Section 9.2 shows example applications in which this feature proved useful.

7 SCnC and CnC Semantics for Keys and Tags

SCnC, because of its streaming nature, differs from general CnC, specifically through the way it accesses data items. In CnC, each item has a unique identifier—its key—which is used to access it, but if items collections are replaced with efficient streaming queues (which are not key-value data structures) the use keys is not possible anymore.

The challenge of translating CnC `get` and `put` keys into stream `push`, `pop` and `peek` lies in converting item keys into streaming offsets. In general CnC, keys can have arbitrary data types, so such a conversion is not possible, but we can leverage the restrictions imposed in step 2 of the analysis phase described in Sect. 5, which ensures that any CnC program that is used with the streaming runtime will only use item keys and control tags with an monotonously increasing integer ordering component.

The conversion between the two types of indexes can be performed by maintaining a current position in the stream, which corresponds to the maximum ordering component encountered so far. If a step performs a `get` on an item with a ordering component larger than the current maximum, the current position in the stream must advance with a number of elements equal to the difference between the two; then, the item with offset zero in the stream is returned. If the step performs a `get` of an item whose ordering component is less than the maximum ordering component, then the item returned is the one at an offset equal to the the difference between the maximum and the current ordering components. Note that access is permitted only to items with ordering components lower than the current position in the stream, which corresponds to a reverse `peek` operation. Traditional `peek` must accomplished by advancing the current position in the stream to the desired position.

This conversion algorithm enables key-based CnC calls to become offset-based SCnC calls in the intermediate representation of SCnC. Thus, the SCnC intermediate representation semantics for an item `get` operation are: if the item access has previously been gotten, `get` is invoked with an offset larger than zero, corresponding to the difference mentioned above. If the next item in the stream needs to be accessed, then `get` with parameter 0, which means that a new item has to be obtained from the stream. `Get(0)` can be performed multiple times to advance the stream one item at a time.

Another difference between the semantics of the intermediate SCnC representation and the CnC semantics is the addition of an optional parameter that controls the dynamic parallelism (see Sect. 6.2). If control tags of the CnC program are bi-dimensional (including both an ordering component and a processing queue id), the processing queue identifier is extracted and used as an extra parameter in control `put` operations.

8 Implementation

In this section we describe our prototype implementation of SCnC, which consists of the SCnC code generator and SCnC runtime. The other phases of he workflow must be performed by hand (these are: well-formed analysis, access pattern analysis, conversion of SCnC `put` and `get` semantics from Sect. 7).

SCnC programs and the SCnC runtime itself are written in Habanero Java (HJ) [10], which offers language primitives for productive and easy to use parallel programming. HJ has asynchronously forked tasks called `asyncs`. Waiting for completion of `asyncs` id done by including them in a blocking `finish` statement. HJ `phasers` [11] are synchronization constructs that enable scalable synchronization between a dynamically variable number of `asyncs`. `Phasers` support both producer-consumer and barrier style synchronization, but we only need producer-consumer synchronisation for SCnC, which we specify be using only the `signal` and `wait` registration modes. `Accumulators` [12] are a reduction construct built over the synchronization capabilities of phasers. An accumulator is associated with a phaser and needs to know the type of the values it is reducing (e.g. `int`) and what is the reduction operation. Each producer task (registered in `signal` mode) performs a `send` to produce a value to be reduced, then `signal` the phaser; when all producers have `signal`ed to the phaser, the consumer tasks (registered in `wait` mode) can access the reduced value by calling the accumulator `result` call.

To implement SCnC, we built an extension of the phaser accumulators called `streaming phasers` that adds support for bounded-buffer synchronization in phasers and accumulators. `Accumulators` now contain an internal circular buffer of size k that is used to store additional items before they are consumed; this enables producer tasks to produce up to k items. Access to previously consumed elements is permitted, within the limits of the internal buffer, by providing an additional offset parameter to the `accumulator.result` call. To enable synchronization on the buffer, phaser producers can now be up to k `signal` operations ahead of the consumers. Accessing previously consumed elements is the key to efficient implementation of the SCnC `get` semantics.

The SCnC runtime consists of implementations for item collections, control collections and step collections. Each item collection has a phaser and accumulator pair that allow synchronization and communication between the producer and consumer of an item collection. Populating the entry control collection from the environment is done through the `init` function of the item collection in which the user performs `put` operations. The SCnC code generator creates a class with these two members and corresponding `put` and `get` operations for each item collection.

As described in Sect. 7, the semantics for an item `get` operation are: if the desire is to access an already used item (an item that has already been gotten at least once), use get with an offset larger than 0. If the intent is to access the next item in the stream, then do a `get` with parameter 0, which means for the runtime that a new item has to be obtained and it inserts the proper wait operation on the item collection phaser. The essential operations of the functions are shown in Listing 1.

```
class ObjectItemCollection {
    public phaser ph;
    public accumulator a;
    public Object get (int no) {
        Object value = null;
        if (no == 0) {
            ph.doWait();
            value = a.result();
        } else {...}
        return value;
    }
    public void put (Object p) {
        a.send(p);
        ph.signal(); }}
```

Listing 1 Code fragment showing the `put` and `get` implementation.

```
void start(WrappedInt tag ) {
  final Tag ftag = tag;
  async phased(
    controlC.ph<WAIT>,
    producedItemC1.ph<SIGNAL> {
    run(ftag);
  }}
void run(WrappedInt ptag) {
  WrappedInt tag = controlC.get(0);
  while (tag.value != endStream) {
    // run the step code
    step(tag);
    // pop the next control tag
    tag = controlC.get(0); }}
```

Listing 2 Code fragment from the step collection base class.

SCnC step collections are modelled through one `async` containing a loop, whose iterations are step instances; the user code for steps is in the `step` function which is called every iteration with a new control tag (see Fig. 2).

9 Experimental Results

The SCnC translator and runtime have been tested on a total of six applications: three StreamIt benchmarks (Beamformer, Filterbank and FMRadio), a clustering application (Facility Location), the well known mathematical algorithm Sieve of Eratosthenes and a video processing application (EdgeDetection). Throughput results are shown in Fig. 7 for four of the applications; because of space constraints, the other benchmarks show only the additional speedup brought by exploiting dynamic parallelism, compared to an execution that fixes at compile-time the amount of data-parallelism exploited by a split-join construct (fixed graph). We report results for CnC (task-based runtime), hand optimized streaming phasers, and SCnC versions of the application.

All applications were initially implemented in CnC. Porting to SCnC by hand helped to validate our algorithm for transforming a CnC spec to SCnC. In this evaluation, we performed the analysis by hand and we manually refactored step code as well: first, changing `puts`s and `gets`s to use the SCnC keys (offsets rather than absolute time iterations), and, second, promoting step-local item collections to local mutable state. The test results have been encouraging, with increases in throughput of up to 40×. In addition, SCnC showed it can support larger problem sizes than CnC due to its improved memory footprint. The input sizes listed for CnC are the largest supported (given a 2GB heap). The SCnC results show the throughput for the same input size as CnC, but also for higher input size that is enabled by using SCnC.

The tests have been performed on a system with four quad-core Xeon processors and 16GB RAM using the methodology by Georges et al. [13]. Our performance analysis focuses on the variation of throughput between implementations and not on absolute timing of applications, as the task-based CnC runtime cannot handle inputs as large as the streaming one. The size of the problem used for CnC is the maximum size

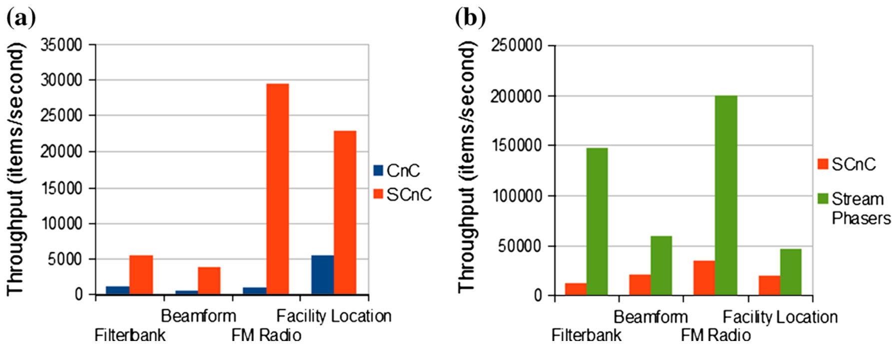

Fig. 7 SCnC performance results. **a** Throughput of SCnC versus CnC at the alrgest input size where CnC does not run out of memory. **b** Throughput of SCnC versus hand tuned streaming phasers at steady rate (large input size)

for which the Java garbage collection does not go over 10 % of application runtime. One thread per core is used for the CnC runtime, which is based on a global work queue SCnC employs one thread per node in the application graph. Therefore graph topology, and in particular the splitting factor at data-parallel split-join nodes, can have an impact on SCnC performance, but not on CnC.

9.1 Benchmarks Without Dynamic Parallelism

This section discusses the performance of the following benchmarks with a statically determined degree of parallelism: BeamFormer, FilterBank, FMRadio and Facility-Location. The first three benchmarks have been ported from the StreamIt benchmarks suite while the last is an implementation of a widely-used problem [14]. On the same input size, SCnC shows an increase in throughput of up to 30× and on larger input size usable only with SCnC (CnC runs out of memory) the throughput increase goes up to 40× (see Table 4).

9.2 Benchmarks with Dynamic Parallelism

The facility location application is a data mining application that solves the problem of optimum placement of production and supply facilities. Formally [14], we are given a *metric space* and a *facility cost* for each node as well as a stream of *demand points*. The problem is finding the minimum number and positioning of nodes such that it minimizes a cost function. This problem occurs in placing production facilities, networking and classification. Our implementation takes advantage of dynamic parallelism by assigning each cluster started by the application to a dedicated processing queue, each with its local state.

In our implementation, it is not possible to get speedup by running multiple consumers in parallel because they are very lightweight and block waiting for input to be produced. This happens because consumers do not compute any of the statistics real

Table 2 SCnC performance with and without dynamic parallelism on the Facility Location benchmark

Work (μs)	Run-time (s)		Speedup
	Static	Dynamic	
0	90	94	0.95
20	212	101	2.1
40	414	131	3.16
80	857	250	3.4

Table 3 SCnC performance with and without dynamic parallelism on the Sieve benchmark, (N = 1,000,000)

Variant	Run-time(s)		Speedup
	Static	Dynamic	
$M = N$	238	40	5.95
$M = 2 * N$	863	80	10.78

implementations of Facility Location would. We modelled this missing computation by additional work times for consumers. The results in Table 2 show an additional speedup of 3.4 compared to the implementation with only static parallelism.

The Sieve of Eratosthenes is a classic algorithm for finding the prime numbers. Our streaming implementation (Fig. 8) has one producer that outputs consecutive numbers starting at 2; the numbers are then sent to several parallel filters that check if the number is divisible with any of the prime numbers that each filter stores. If a filter finds a divisor, it sends to the join node a 1, if not, it sends 0. The join step performs a sum-reduction and if the result is 0, the number is prime. It then sends back to the filters the id of the queue that should add the newly discovered prime number to its prime number store and the cycle repeats itself.

The number of processing queues is not application-dependent and can be adjusted to match the number of cores in the machine. Each place stores a chunk of the primes previously found. Performance results are found in Table 3 for 15 filters and a cyclic distribution of prime numbers to filters.

We also implemented an extension of the Sieve that not only finds the prime numbers up to N, but also counts the numbers between N and M that are not divisible by any prime number less than N. With it we analyze the speedup obtained by dynamic parallelism without the overhead of variable granularity and added feedback synchronization as shown in Table 3.

Our implementation of image edge detection application [15–17]relies on detector filters that rotate the image a certain angle and then apply a one dimensional convolution kernel. Because the convolution detects edges on a single dimension, each detector uses a different angle of rotation. Then, a join overlaps the result of the detectors and sums all the pixels considered as edges. This sum is used to increase or decrease the number of detectors (angles) used for the next frame, in order to balance accuracy and speed (see Fig. 8).

For benchmarking, the number of edge detection filters was fixed to 2, 4 and 8, although in practice it varies frame to frame, so that we can analyze the relationship

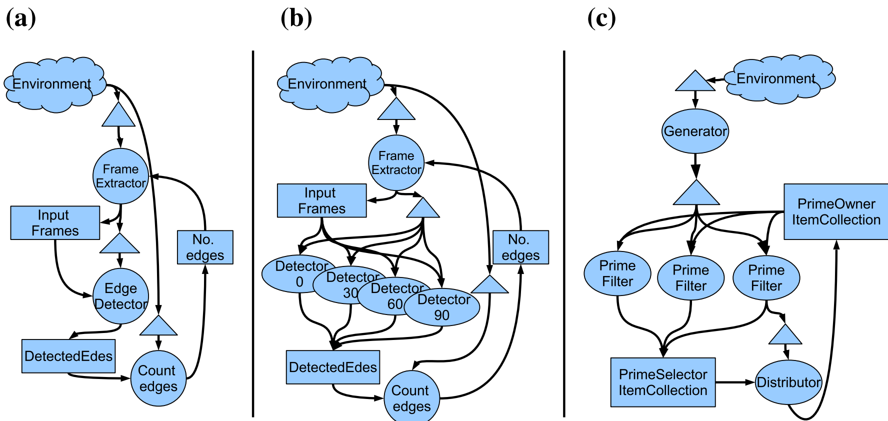

Fig. 8 **a** The CnC graph of Edge detection. **b** The dynamic behavior of SCnC, using 4 processing queues for Detector. **c** The dynamic behavior of Sieve, with 3 processing queues for the PrimeFilter

Table 4 Edge detection results

No. detectors	Throughput (items/second)		
	Streaming phasers	SCnC	CnC
2	14.3	14.3	15.8
4	2.63	2.38	1.88
8	1.25	1.15	0.61

between execution time and the number of cores used. The relatively small speedup of SCnC compared to CnC for this application (see Table 4) is explained by the higher granularity of the steps (each detector processes an full input frame). The time consumed by the communication and synchronization, be it for a task-based or streaming runtime, is very small in comparison with the work performed by the steps.

10 Related Work

Compared to other streaming systems, SCnC is of medium generality in the graph shapes it supports. StreamIt [1] only allows split/join/loopback patterns. The Lime project[18] allows loopback streams but only to destinations upstream from the source, disallowing streams to filters that are not on the path from the source to the root (input node) of the streaming graph. However, there are many other system which impose fewer restrictions than SCnC on the graph shape.

One reason for the more restrictive nature of StreamIt compared to SCnC is their bias towards DSP stream applications, which are run on systems with high throughput, but low flexibility and having a more flexible programming model would not only be useless, but also would make optimizations more difficult, lowering performance. On the other hand, we target multicore architectures where taking advantage of streaming

for a larger class of programs would bring considerable performance improvement, to more applications.

Brook [19,20] takes a different approach to managing granularity: if StreamIt uses fission and fusion to get to a steady schedule starting from fine grained operations, Brook exposes only coarse grained multi-dimensional data structures (called streams) to the programmer who is expected to process them through predefined operators. Using stream shape analysis they end up performing kernel fusion and optimizations similar to loop interchange. They target both multiprocessors and GPU systems. Because SCnC relies instead of CnC as a base language, SCnC steams can be as coarse or fine grained as the programmer desires and dynamic splitting can be used to achieve finer granularity.

The problem of finding a better task mapping for stream programs has been tackled for years [21]. Static mapping, together with dynamic adjustments for load balancing have been implemented and shown good performance [22]. Recently, dynamic approaches have become possible because of work performed in the execution time prediction for streaming tasks. The dynamic parallelism approach we propose comes to complement several projects that aim to offer tailored load balancing for streaming applications Farhana Aleen et al. [23] use taint analysis and simulation to identify pipeline delays as a function of input data. They could profit from adding dynamic split-joins to complement their analysis to get dynamic load balancing optimization.

A comparison between the task-based dataflow model and streaming is started by Miranda et al. [24], but their work relies on special language and the comparison with data-flow is not tweaked for performance, relying on the general Cilk model for short-lived tasks. The paper uses a single and synthetic benchmark and shows mixed results. On the other hand, SCnC consistently outperforms CnC for a larger number of applications, even without using a custom-built streaming language, compiler and intermediate representation and while maintaining determinism. It only has minor changes relative to the CnC variant.

Vandierendonck at al. [25] propose a combined runtime that offers both the guarantees of work stealing and the ease of use of dataflow. SCnC is the natural step forward towards task parallelism + X through its application on the streaming domain, its identification of streaming patters and use of a single input language.

The work most similar to our approach of turning streaming into an automatic optimization for task based programs is OpenStream [26]. The project proposes dataflow extensions for OpenMP to automate the compilation of general dataflow programs (as illustrated by StarSS) to the OpenStream model. Instead of a traditional streaming runtime such as SCnC, OpenStream relies only on a point to point dependence resolution and as such is complementary to our work. OpenStream is the only published system, other than SCnC, that can handle stateful dynamically parallel filters.

11 Conclusion and Future Work

This paper proposes algorithms that turn streaming into an automatic optimization that can be applied to applications written for task-based execution. We showed how to formally define and identify the graph shapes and access patterns that characterize

streaming applications. We ensure that the streaming execution and task-parallel execution are deadlock-equivalent by automatically sizing the streaming buffers such that there are no streaming-induces deadlocks. We identified optimizations that improve performance from applications written for task parallelism, but are instead ran using the streaming parallelism, though stateful dynamic parallelism.

The experimental results show that the model offers significant performance improvement when compared to the task-parallel approach of running applications, with and without dynamic parallelism.

Future directions of research consist of integrating SCnC with the task-parallel CnC runtime, so that the system can stream application subgraphs when streaming the complete application would be incorrect. Also, our dynamic parallelism approach has a natural extension in dynamic pipelines for a more flexible dynamic parallelism support.

References

1. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: a language for streaming applications. In: CC '02, pp. 179–196. Springer, London
2. Budimlic, Z., Burke, M., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D.M., Sarkar, V., Schlimbach, F., Tasirlar, S.: Concurrent collections. Sci. Program. **18**(3–4), 203–207 (2010)
3. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**, 720–748 (1999)
4. Agarwal, S., Barik, R., Bonachea, D., Sarkar, V., Shyamasundar, R.K., Yelick, K.: Deadlock-free scheduling of X10 computations with bounded resources. In: SPAA '07 ACM, New York
5. Guo, Y., Barik, R., Raman, R., Sarkar, V.: Work-first and help-first scheduling policies for async-finish task parallelism. In: IPDPS'09
6. MathWorks Symbolic Math Toolbox Documentation. http://www.mathworks.com/help/symbolic/index.html. Accessed Feb 2015
7. Li, P., Agrawal, K., Buhler, J., Chamberlain, R.D.: Deadlock avoidance for streaming computations with filtering. In: SPAA '10
8. Li, P., Agrawal, K., Buhler, J., Chamberlain, R.D., Lancaster, J.M.: Deadlock-avoidance for streaming applications with split-join structure: two case studies. In: ASAP, pp. 333–336 (2010)
9. Soul, R., Gordon, M.I., Amarasinghe, S., Grimm, R., Hirzel, M.: Hitting the Sweet Spot for Streaming Languages. NY University CS Technical Report TR2012-948 (2009)
10. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-java: the new adventures of old X10. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ '11 (2011)
11. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In: ICS '08, pp. 277–288, ACM, New York
12. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phaser accumulators: a new reduction construct. In: IPDPS 09
13. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. In: OOPSLA'07, pp. 57–76. ACM
14. Meyerson, A.: Online facility location. In: FOCS '01
15. Canny, J.: A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **8**, 679–698 (1986)
16. Nijhuis, M., Bos, H., Bal, H.E.:A component-based coordination language for efficient reconfigurable streaming applications. In: ICPP (2007)
17. Nijhuis, M.: Framework for parallel streaming applications. Ph.D. dissertation (2007)
18. Auerbach, J., Bacon, D.F., Cheng, P., Rabbah, R.: Lime: a java-compatible and synthesizable language for heterogeneous architectures. In: OOPSLA '10, pp. 89–108, ACM, New York
19. Liao, S., Du, Z., Wu, G., Lueh, G.-Y.: Data and computation transformations for brook streaming applications on multiprocessors. In: CGO '06, pp. 196–207, IEEE Computer Society, Washington

20. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for gpus: stream computing on graphics hardware. In: SIGGRAPH '04, pp. 777–786, ACM, New York (2004)
21. Aoyagi, Y., Uehara, M., Mori, H.: A case study on predictive method of task allocation in stream-based computing. In: Proceedings of the 13th International Conference on Information Networking, ICOIN '98
22. Collins, R.L., Carloni, L.P.: Flexible filters: load balancing through backpressure for stream programs. In: EMSOFT '09
23. Aleen, F., Sharif, M., Pande, S.: Input-driven dynamic execution prediction of streaming applications. In: PPoPP '10, pp. 315–324
24. Miranda, C., Pop, A., Dumont, P., Cohen, A., Duranton, M.: Erbium: a deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes. In: CASES '10, pp. 11–20. ACM
25. Vandierendonck, H., Tzenakis, G., Nikolopoulos, D.S.: A unified scheduler for recursive and task dataflow parallelism. In: PACT '11
26. Pop, A., Cohen, A.: Openstream: expressiveness and data-flow compilation of openmp streaming programs. In: TACO '13